SETTING UP AN AUTONOMOUS MULTI-UAS LABORATORY: CHALLENGES AND RECOMMENDATIONS

by

Nadia M. Coleman

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Electrical and Computer Engineering

School of Electrical and Computer Engineering

West Lafayette, Indiana

May 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Shreyas Sundaram,

Department of Electrical and Computer Engineering

Dr. David Cappelleri,

Department of Mechanical Engineering

Dr. Jianghai Hu,

Department of Electrical and Computer Engineering

Dr. Nicolò Michelusi,

Department of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the Graduate Program

ACKNOWLEDGMENTS

I'd like to thank Daniel McArthur for his assistance with my thesis work, and also Amritha Prasad for providing the task allocation algorithm implemented in this thesis. I wish to gratefully acknowledge my thesis committee for their insightful comments and guidance and my family for their support and encouragement. I dedicate this thesis to my late friend, Greg Brunson, whose memory continues to inspire me and push me forward.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

UAS	unmanned aerial system
MAV	micro air vehicle
ROS	Robot Operating System
FCU	flight controller unit
IP	internet protocol
DHCP	dynamic host configuration protocol
LAN	local area network
MAC	media access control
OS	operating system
USB	universal serial bus
LTS	long-term support
URI	uniform resource identifier
QGC	QGroundControl
FC	flight controller
RTF	ready to fly
URL	uniform resource locator
UDP	user datagram protocol
TCP	transmission control protocol
EKF	Extended Kalman Filter
HAPP	Heterogeneous Agent Path Problem
SITL	software in the loop
SSH	secure socket shell

GLOSSARY

Unmanned Aerial System – An aircraft without a human pilot on board and a type of unmanned vehicle

Robot Operating System – A flexible framework for writing robot software including a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

GitHub – a Git repository hosting service.

Dronecode – A nonprofit hosted under the Linux Foundation, dedicated to fostering open-source components and their communities.

PX4 Autopilot – An open-source BSD-Licensed flight control software for drones and other unmanned vehicles, a project by Dronecode.

MAVLink – An open-source lightweight communication protocol for UAV systems and components, widely used for communications between ground-stations, autopilots and companion computers, a project by Dronecode.

QGroundControl – An open-source feature complete and fully customizable control station for MAVLink based Drones, a project by Dronecode.

MAVROS – A MAVLink extendable communication node for ROS with proxy for Ground Control Station.

Gazebo Simulator – A ROS compatible open-source 3D robotics simulator.

EKF2 Estimation System – An attitude and position estimator using an Extended Kalman Filter.

ABSTRACT

There is a significant amount of ongoing research on developing multi-agent algorithms for mobile robots. Moving those algorithms beyond simulation and into the real world requires multi-robot testbeds. However, there is currently no easily accessible source of information for guiding the creation of such a testbed. In this thesis, we describe the process of creating a testbed at Purdue University involving a set of unmanned aerial vehicles (UAVs). We discuss the components of the testbed, including the software that is used to interface with the UAVs. We also describe the challenges that we faced during the setup process, and evaluate the UAV platforms that we are using. Finally, we demonstrate the implementation of a multi-agent task allocation algorithm on our testbed.

CHAPTER 1. INTRODUCTION

1.1 Unmanned Aerial Systems Research

The development of intelligent multiple unmanned aerial systems (multi-UAS) has been at the forefront of system and controls research. Various applications of multi-UAS research are driven by military and defense motivations [1–3], while others pertain to environmental monitoring and disaster recovery [4–6]. A particularly relevant application in the midst of the COVID pandemic of 2019-2020 is the use of UAS to support social-distancing practices (through industrial and commercial delivery drones) [7–9]. However, as the applications of multi-UAS systems become increasingly complex, they require more sophisticated multi-agent algorithms which must be transitioned from simulations and theory into practice. This calls for the creation of multi-UAS laboratories which will facilitate the refinement and optimization of multi-agent systems for final deployment.

This work describes the process of establishing a multi-UAS laboratory. First we present a catalog of necessary equipment and setup instructions. Then we build the foundations for autonomous programming in ROS. Finally, we use our laboratory to implement an algorithm for a heterogeneous multi-agent path planning scenario. Alongside this thesis, we offer a repository in GitHub that includes detailed instructions and example programs [10]. In this thesis we set forth a guide along with recommendations for constructing a reliable and extensible multi-UAS laboratory so that we may foster the growth of distributed controls research by equipping prospective researchers with the tools they need to assemble their own laboratories.

1.2 Existing Laboratories

There are multi-UAS laboratories located at various universities worldwide including the University of New Mexico's Multi-Agent Robotics and Heterogeneous Systems Laboratory (MARHES) [11], and Stanford's Multi-Robot Systems Laboratory [12]. These laboratories and their staff contribute valuable insights, usually in the form of publications, that continually form a basis for further research into multi-UAS algorithms. Multi-agent testbeds extend the study of algorithms by testing in real world conditions not observable by simulations.

If the means to build a lab are unattainable, additional options for multi-agent experimentation exist. An open access multi-agent testbed, called the Robotarium, is a new concept being implemented by Georgia Tech [13]. The Robotarium is a multi-robot laboratory that can be remotely accessed and used by researchers to implement their own algorithms. This allows users the benefits of testing with physical robots without having to build a testbed. However, the Robotarium is limited to a set of homogeneous, planar (2D) robots called GRITSBot X, thus heterogeneous, 3D multi-UAS experiments cannot be performed on this platform. Also, the use of the Robotarium is subject to operating times and availability depending on the current demand.

Building a multi-UAS laboratory is favorable because you may perform all manner of experiments including 2D, 3D, homogeneous, heterogeneous, etc. Owning a lab also has the advantage of real time debugging and optimization during experiments. Another advantage is the inherent choice of which robots to experiment with and the option to innovate new robots. Ownership of a multi-UAS laboratory gives researchers full control and creative liberty when developing and conducting experiments.

1.3 Tools for Drone Research

Here, we introduce open source drone research tools widely used in the drone research community.

1.3.1 ROS

The Robot Operating System (ROS) is an open-source, modular set of software libraries and tools that help users build robot applications, including multi-UAS applications [14]. ROS hosts a collection of pre-developed ROS packages for robotic applications and allows users to create custom packages. A ROS program consists of "publisher nodes" that publish messages of various types over ROS topics (named buses over which nodes exchange messages) to "subscriber nodes" which convert the messages as needed.

1.3.2 Dronecode Research Tools

Dronecode, a nonprofit organization, has developed an abundance of open-source tools for drone research including PX4 Autopilot, MAVLink, QGroundControl(QGC), and MAVSDK [15]. PX4 Autopilot is an open-source BSD-Licensed flight control software that delivers guidance, navigation, and control algorithms for various air-frames and incorporates altitude and position estimation [16]. MAVLink is an open-source lightweight communication protocol for UAS systems and components, widely used for communications between ground-stations, autopilots, and companion computers [17]. QGroundControl is a ground-station application that provides status and flight control configuration information for MAVLink enabled drones [18]. The ROS package, MAVROS, is a MAVLink extendable communication node for ROS with proxy for a ground control station such as QGroundControl [19]. We will leverage these tools to build our multi-UAS testbed for the study of multi-agent algorithms.

1.3.3 Gazebo

Simulation tools are an important resource for testing multi-agent robotics systems rapidly. Simulations offer insight into potential behaviors as well as safety. One such tool is Gazebo, an open source 3D robotics simulator equipped with a robust physics engine, high-quality graphics, and convenient interfaces [20]. Gazebo is especially configured for use with ROS, allowing users to effortlessly test the same source code they would for a physical system, and derive conclusions about efficiency and how robots will occupy a physical space, as well as make predictions about efficiency and robustness of a test when applied to a physical system. Being open source, implementation with Gazebo is easy due to the substantial amount of resources provided by the Gazebo community.

Although Gazebo is a very useful tool, it (like any other simulator) has its limitations. The source installation of Gazebo offers various models of commonly seen objects such as simple 3D geometric objects, traffic barriers, furniture, and even whole buildings. The list of models also includes a limited selection of well known robotic vehicles such as the Iris quadrotor or Turtlebot ground robot. If testing with one of these specific models, one can potentially expect a more realistic simulation as compared to the physical system; however, working with new models can be challenging. Although Gazebo's user interface is fairly well developed, creating a new model file is not a trivial task for even an intermediate user. Fortunately the use of the Iris quadcopter model was sufficient for our purposes given that it is a PX4 Autopilot based vehicle like our two physical vehicles, the Intel Aero RTF quadcopter and the Uvify Draco-R hexacopter. It should also be noted that simulating all the possible conditions in nature can never be perfected; however Gazebo's physics engine is state of the art, making it the premier robotics simulator for the robotics community [21].

CHAPTER 2. LAB EQUIPMENT SETUP

In this chapter, we will discuss the equipment, setup, and configuration necessary for building a multi-UAS laboratory.

2.1 Assembling a Laboratory Space

An essential requirement for a multi-UAS testbed is a lab with ample space to operate in. Our lab, the Perception-Based Engineering Lab, located in Herrick Laboratories at Purdue University is approximately 13.1 meters wide, 8.5 meters long, and 6.7 meters high as seen in Figure 2.1. For safety a net curtain was installed that surrounds the entire flying space. Typically the netted area should be clear of all objects with the exception of objects being used to perform experiments. Our lab has a control room area outside of the netted area that is used as an observation and communications center for performing tests.

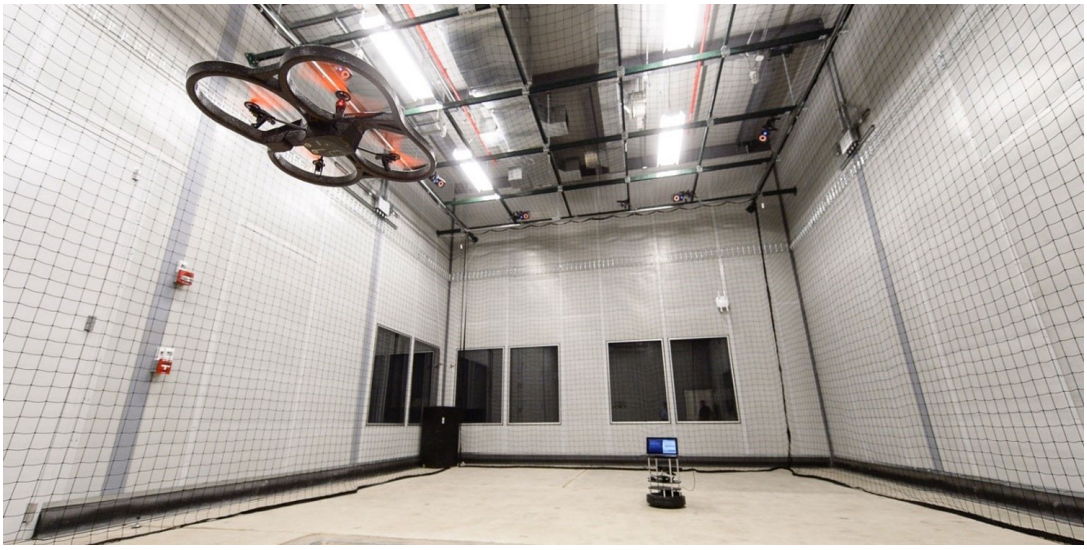

Figure 2.1. Perception-Based Engineering Laboratory, from Purdue Engineering [22]

2.2 Wireless Router Setup

A wireless router is needed to provide a communication link between all of the equipment we will need for our multi-UAS lab setup. It is desirable that all computing equipment is uniquely recognized by a static Internet Protocol (IP) address which can be achieved by configuring a Dynamic Host Configuration Protocol (DHCP) server in the router settings (typically accessible by typing in the router's IP address into a web browser). Once in the settings, we enable the DHCP server, choose a domain name, and then choose our IP pool starting and ending address. The total pool size of a DHCP server supports only 254 addresses so in order to reserve the maximum number of addresses (assuming the machine's IP is 192.168.1.1) we can set the IP pool starting and ending addresses to 192.168.1.2 and 192.168.1.254, respectively. Finally manual assignment should be enabled in order to store and assign the IP address of any device that connects to the network.

Once this is enabled and a device is connected, its Media Access Control (MAC) address and IP address will be visible, which can be altered to any unassigned IP address between the pool starting and ending addresses. One can then save, edit, or delete the device, MAC address, and assigned IP by selecting add/delete. One can verify that the device has correctly been assigned the chosen IP by connecting it to the network and viewing its IP address using the terminal command 'ifconfig' or 'ipconfig' depending on the device.

2.3 Vicon Motion Capture Setup

2.3.1 Vicon System Setup

In order to implement autonomy, each drone must have a way to localize itself in the flying area, i.e., a drone must know where it is in order to figure out where it must go. To accomplish this, we can incorporate a motion capture system into the lab space. A motion capture system is set of sensors and equipment used to visualize and capture position data. Typically, this is accomplished using infrared cameras that visualize the positions of reflective markers arranged in a specific form over time. Our lab is equipped with twelve Vicon T160 motion tracking cameras positioned uniquely around the top frame of the netted flying area. Any Vicon system purchase includes manuals detailing setup instructions which we will not cover here.

2.3.2 Vicon Desktop Computer Setup

Once our motion capture system is installed, we proceed by installing the motion capture software on our desired computer. Our lab utilizes a computer with Windows Vista located in the lab's control room and has Vicon's Nexus and Tracker programs installed. The software comes with the purchase of the motion capture system along with installation instructions and thus we will not cover those here. Once the necessary software is installed, one can verify that the motion capture system is functioning using the relevant software program(s).

2.4 Base Station Desktop Computer Setup

Given that the lab space is adequately prepared for testing and the motion capture system is up and running, we proceed to establish a base station computer.

2.4.1 Operating System

We highly recommend using the Ubuntu operating system (OS) given it is the only OS supported by ROS. We also suggest using the latest long-term supported (LTS) desktop version of Ubuntu (although any version can be used as long as it is classified as LTS). Non-LTS Ubuntu versions are revised every 6 months, as opposed to LTS versions which are released every 2 years, making the non-LTS versions less stable and more prone to bugs. Also, it is important to be aware that the version of Ubuntu will coincide with the version of ROS that will be used. We chose to install Ubuntu desktop version 18.04.3 LTS. We then connect the machine to our wireless network and access the router settings to add the device to the DHCP server and assign a static IP address.

2.4.2 ROS

Once Ubuntu is installed and ready, we install the version of ROS that's compatible with the version of Ubuntu used, which in our case was ROS Melodic. Configure ROS by setting the environment variables to the appropriate value which, if the tests are to be run locally, depends on the machine's IP address. These values may be set in the machine's '.bashrc' file by adding some simple lines of code: 'export ROS_MASTER_URI = http://COMPUTERS_IP:11311' and 'export ROS_IP = COMPUTERS_IP'. Now we can take advantage of ROS's extensive libraries or create our own.

2.4.3 QGroundControl and Gazebo

We also installed Dronecode's open-source ground control station program, QGroundControl (QGC) which, once connected to a MAVLINK enabled drone, reports essential status and parameter information such as battery level, flight mode, etc.

Most importantly QGC allows us to view and modify all of the PX4 flight controller parameters that dictate the many features of the drone. QGroundControl is widely used by the drone research community and is backed by a forum community where one can find answers to technical difficulties or learn more about how command messages move though the PX4 flight stack to the hardware [23]. We chose to use the Gazebo 3-D simulation tool alongside our physical experiments and installed the necessary Gazebo ROS tools on our base station Ubuntu machine.

2.5 Optional Laptop Setup

We installed QGC on a Mac OS laptop and added the device to the DHCP server. This laptop was used to monitor battery consumption during experiments and also to send corrective flight commands in case of emergencies. Incorporating a ground station laptop is optional as you may choose to run QGC on the base station computer instead; however QGC should be easily accessible when running tests in case of emergencies (to avoid frantically switching between computer programs to get to QGC).

2.6 UAS Platform Setup

2.6.1 Selecting a UAS for Research

There are a multitude of features to consider when choosing a UAS platform suitable for experimentation purposes. Here we will discuss the criteria we considered when choosing our UAS platforms.

There are a variety of commercial and custom platforms available. In order to determine our criteria, we considered the nature of the experiments we intended to perform. Generally, we intended to utilize ROS conjointly with the PX4 flight stack to fly multiple vehicles simultaneously in an autonomous offboard fashion.

The most important criterion was our desire for the platform to be PX4 autopilot and research based as such platforms are preconfigured to be integrated into research testbeds. Also, given that the PX4 autopilot is an open source software, the software, its tools, and applications are well supported and widely documented by the drone research community.

Another criterion for choosing any UAS platform is cost. Conducting a multiple vehicle experiment meant we had to purchase multiple vehicles, with a corresponding increase in purchase price.

Next we evaluated the ideal size of a platform given that the flying volume of the lab is limited; the smaller the platform the more we can fit into our limited lab volume at one time. This meant that vehicles like octocopters or fixed wing UAVs would require too much space for our applications.

Based on these criteria, we landed on two research-based platforms: the Intel Aero RTF quadcopter [24] (Figure 2.2), and the UVify Draco-R hexacopter [25] (Figure 2.3). The Intel Aero platform was an ideal candidate given that it was a fully assembled ready to fly drone that required minimal adaptation for use with ROS, and at a fair price (approximately $1,100 each). In terms of size the quadrotor sits at 360 mm hub-to-hub, with a weight of 865 grams without the battery which met the requirements for our purposes. The Intel Aero platform also had ample documentation both in the form of repositories and community support forums where other drone researchers shared their experiences. However, after our initial purchase of two Intel Aero drones, the platform was discontinued by Intel. Fortunately, many of its applications are extendable to other platforms.

The UVify Draco-R platform was suitable as it came preconfigured for use with ROS. The Draco-R came preloaded with the latest at the time PX4 autopilot and ROS enabled Ubuntu companion computer. Although this drone has six rotors it is still remarkably small with a hub-to-hub length of 370mm, making it just slightly larger diagonally than the Intel Aero quadcopter. This platform did not have the same breadth of resources as the Intel Aero however we were able to reach out to UVify for support when needed. The Draco-R was also more expensive than the Intel Aero (approximately $4,285 each). With minimal adjustments the Draco-R was ready to be deployed on our multi-UAS testbed.

Figure 2.2. Intel Aero from [26]

2.6.2 Intel Aero Setup

The two Intel Aero drones came ready to fly so the first step was to test fly them with the receiver which was successful. We then followed the initial setup instructions by Intel which includes basic operation information as well as how to get the Aero connected to QGroundControl and how to calibrate the sensors [27].

The Aero comes with Linux Yocto OS installed (which is not a supported OS for using ROS) so our next step was to follow guidance distributed by Intel for installing Ubuntu [28]. Detailed information on installing Ubuntu and troubleshooting can be found at the Autonomous Multi-UAS Laboratory GitHub repository that we have posted online [10]. Initially the Aero will emit a local wireless hot-spot that enables QGC. However this hot-spot will be deleted once Ubuntu is installed, so MAVLink router must be configured in order to connect to QGC. Instructions for configuration are included in the instructions by Intel [28], as well as in the repository we have posted [10]. Once Ubuntu is installed, the sensors will need to be re-calibrated using QGC. Upon completion of these steps we performed a manual test flight to ensure the Aero was still functioning properly with the updates we installed.

Next, we install and configure ROS on the Aero. ROS has easy to follow instructions on their website [29]. Our setup includes a base station computer that will be responsible for serving as the ROS master node. In order for the drone to recognize the base station computer as the ROS master (the component which enables individual ROS nodes to locate one another) the following environment variables must be included in the drone's '.bashrc' file by adding some simple lines of code: 'export ROS_MASTER_URI = http://BASE_STATION_COMPUTERS_IP:11311' and 'export ROS_IP = DRONES_IP'. However if roscore is to be started on the drone itself, one should use: 'export ROS_MASTER_URI = http://DRONES_IP:11311'.

We also chose to update the PX4 flight controller firmware version in order to use the newer features. The detailed process is described in the repository [10].

Figure 2.3. UVify Draco-R from [30]

2.6.3 UVify Draco-R Setup

The Draco-R arrived with Ubuntu, ROS, and PX4 pre-installed which expedited the setup process. First we performed a manual flight test on the two Draco-R platforms to verify they were functioning properly. Next we installed the MAVLink router, a tool made by Intel which routes packets between UDP or TCP endpoints (e.g., between the vehicle hardware running the PX4 flight stack and QGroundControl) to stream status and parameter information [31].

2.6.4 Configuring PX4 Parameters for Multi-Vehicle Experiments

In order to proceed with experiments we must carefully configure the UAS flight controller unit (FCU) parameters. This extensive list of parameters dictates every feature of the UAS, including sensor calibration, position estimator type, safety configurations, and more. A full parameter list can be found on Dronecode PX4 website [32]. We can view and modify these parameters in QGroundControl. Table 2.1 indicates the parameters we changed and related information. The commander and mission parameters were altered to account for our desired safety features. The EKF2 parameters were changed to configure the data fusion features of the FCU. The MAVLink MAV_SYS_ID parameter is set to discern one vehicle from another and should be different for every vehicle that is deployed.

It is immensely important to proceed with caution when changing FCU parameters. In one instance during parameter configuration of an Intel Aero drone, we changed the MAVLink MAV_COMP_ID parameter from its default value which caused us to lose connection to QGC and so prevented us from being able to undo the parameter change. Our first attempt at resolution was to reset the FCU by re-flashing the PX4 firmware, however we could not regain connection to QGC. A second attempt to fix this issue was made by creating a serial device communication link which also did not succeed. We were left to believe that the FCU hardware had failed.

In a similar instance on a Draco-R drone we changed the MAVLink MAV_1_CONFIG parameter from 102 (TELEM2) to 6 (UART 6) which also caused us to lose connection to QGC. We were able to resolve this issue by creating a serial device communication link so that we could connect to the FCU through a USB cable allowing us to regain access to the parameters. Once the parameter change was undone we regained connection to QGC over wifi. The parameter settings shown in Table 2.1 should be consistent for every drone and will allow for safe multi-UAS experimentation.

Table 2.1.: Table of Configured FCU Parameters

Type	Name	Description	Default Value	New Value
Commander	COM_ARM_EKF_AB	Max value of EKF accelerometer delta velocity bias estimate that will allow arming.	1.73e-3 m/s	0.0024 m/s
	COM_DISARM_LAND	Time-out for auto disarm after landing	-1 s	0 s
	COM_DL_LOSS_T	Datalink loss time threshold	10 s	5 s
EKF2	EKF2_AID_MASK	Integer bitmask controlling data fusion and aiding methods	1	24
	EKF2_GPS_CHECK	Integer bitmask controlling GPS checks	245	21
	EKF2_HGT_MODE	Determines the primary source of height data used by the EKF	0: Barometric pressure	3: Vision
	EKF2_REQ_EPH	Required EPH to use GPS	3.0 m	5.0 m
	EKF2_REQ_EPV	Required EPV to use GPS	5.0 m	8.0 m
	EKF2_REQ_HDRIFT	Maximum horizontal drift speed to use GPS	0.1 m/s	0.30 m/s
	EKF2_REQ_SACC	Required speed accuracy to use GPS	0.5 m/s	1.00 m/s
	EKF2_REQ_VDRIFT	Maximum vertical drift speed to use GPS	0.2 m/s	0.50 m/s
MAVLink	MAV_SYS_ID	MAVLink system ID	1	VEHICLE #
Mission	COM_OBL_RC_ACT	Set offboard loss failsafe mode when RC is available	0: Position mode	4: Land mode
	MIS_TAKEOFF_ALT	Take-off altitude	2.5 m	1.0 m
	NAV_ACC_RAD	Acceptance Radius	10.0 m	2.0 m
	NAV_DLL_ACT	Set data link loss failsafe mode	0: Disabled	3: Land mode
	NAV_RCL_ACT	Set RC loss failsafe mode	2: Return mode	0: Disabled

CHAPTER 3. SETTING UP SOFTWARE INFRASTRUCTURE FOR MULTI-UAS PROGRAMMING

In this chapter, we will discuss the requirements for autonomous multi-UAS programming in ROS.

3.1 Autonomous Programming in ROS

We created a custom ROS package on our base station computer that includes all of the key files for multi-vehicle experimentation, including all the necessary ROS environment scripts. The process for creating this package is further described in this section.

3.1.1 ROS Packages

A ROS package commonly hosts four types of files including header files (.h), source files (.c, .cpp), launch files (.launch), and scripts (.py) [33]. These files are saved in their four respective directories within a ROS workspace which are named, “include,” “src,” “launch,” and “scripts.”

3.1.2 Publishing Vicon Data in ROS

A basis for publishing Vicon position data is written in C++ and includes a header file (.h), and a source file (.cpp), which are saved in the “include,” and “src” folders respectively. In order to publish this data in ROS we created a ROS launch file with a matching filename which must be written in a specific XML format. The Vicon system we have in our multi-UAS laboratory came with example scripts for publishing position data over rostopics in the form of PoseStamped ROS messages which we used as the basis for our implementation. Due to copyright we unable to post them in the Autonomous Multi-UAS Laboratory repository [10].

3.1.3 Autonomous Programming

Autonomous programming in ROS can be accomplished using either C++ or Python. There are an abundance of resources and tutorials for autonomous programming for both languages. Also, we have posted all of our scripts in our GitHub repository for reference [10]. For our purposes we chose to write our autonomous scripts in Python. Python scripts are saved in the "scripts" directory of the ROS workspace and can be run using the "./filename.py" command in the terminal.

3.1.4 Configuring MAVROS

In order to perform multi-UAS experiments, MAVROS must be configured for each UAS to be operated simultaneously. This is true both for simulation in Gazebo as well as in physical experiments. The pertinent MAVROS configurations must be made in a launch file. A group name-space tag can be used to append a unique name to each vehicle. This group name will append to the front of all associated MAVROS rostopics; for example the MAVROS state rostopic "/mavros/state" will now appear as "group_ns/mavros/state" for each unique name-space. Each vehicle must also be assigned a unique system ID (system_id) and target system (tgt_system_id) ID. Typically the target component ID should not be changed from the default value. Configuring these unique vehicle identification values will ensure that the correct instructions are published to the correct vehicle.

3.1.5 Multi-Agent Simulation using Gazebo

A tutorial for multi-vehicle simulation is readily available at the Dronecode website [34]. There are some nuances to getting multiple vehicles running, specifically getting the proper addressing correct. Each vehicle has a list of flight control parameters that dictate how the autopilot is configured. The key parameters of interest that must be unique for each vehicle to be operated in tangent are the MAVlink system ID (MAV_SYS_ID), system ID (system_id), and target system ID (tgt_system) as discussed in Section 3.1.4. The MAVlink system ID should be uniquely configured as discussed in Section 2.6.4.

In terms of addressing, there are also some parameters that must be unique to each vehicle including the FCU URL and MAVlink UDP port. The FCU URL corresponds to the connection between the UDP port of the API/Offboard and the UDP port of the ground control station (QGroundControl) and uses the IP of the computer running the simulation. The FCU URL should be uniquely assigned using the form "14540+(vehicle_id#)@localhost:14550+(vehicle_id#)" (e.g, the FCU URL for UAV1 should be "14541@localhost:14551" and for UAV2 "14542@localhost:14552", etc.). The MAVlink UDP port is the UDP port of the Simulator. The MAVLink UDP port should follow the form 14560+(vehicle_id#) (e.g., the MAVlink UDP values of UAV1 and UAV2 should be set as 14561 and 14562, respectively). An example launch file is provided in our git repository, showing how these parameters are configured for two vehicles [10]. A simple MAVlink communications chart with address information can be seen in Figure 3.1. Other in depth information on UDP port configuration for simulation can be found on the PX4 website [35].

In concert with the above configurations, the values for each individual vehicle model configuration file must match. For example if there are two drones modeled by the iris quadcopter, there should be two configuration files called "iris_1" and "iris_2". The parameter "SITL_UDP_PRT", and MAVlink start and stream information must mirror the configurations in the launch file. Examples of these configuration files can be found in our git repository [10].

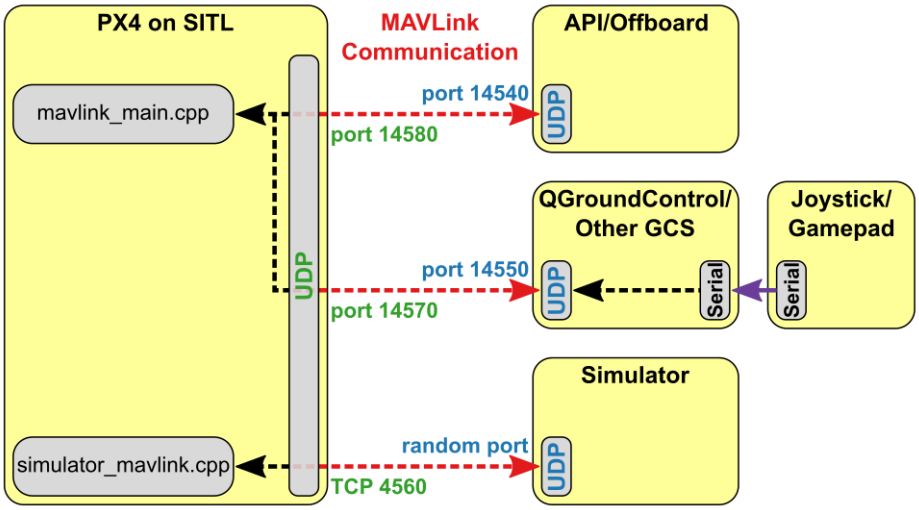

Figure 3.1. MAVLink Communication, from Dronecode [36]

3.2 Multi-Vehicle Experimentation in ROS

With all of the previous measures in place we are ready to implement experiments in our multi-UAS laboratory. We start up the Vicon to allow the cameras to warm up and open the necessary software, in our case Nexus, and note the IP address of the machine running this software. We make sure we have fully charged batteries for all the drones we plan to experiment with. We inspect each drone to ensure they are not damaged or require any maintenance. We then connect to power, securely attach the propellers if needed, and connect any other requisite peripherals. Then we start the MAVLink router and verify that we have communication with QGC. This will also start a new ULog file which logs any and all system and state data. We can verify that the batteries are fully charged and there are no preflight check errors in QGC. On the base station computer, we open a terminal and start ROS by running the terminal command "roscore" which initializes all of the pre-requisite ROS nodes and programs. This can also be done by running "roslaunch [filename].launch". Next we begin publishing the Vicon position data in ROS over the "/mavros/vision_pose/pose" rostopic by running "roslaunch vicon_stream_multi.launch ip=:VICON_DESKTOP_IP". We then log into each drone remotely by SSH and start a MAVROS node natively using "roslaunch msral mavros.launch". Finally our system is ready for us to run our autonomous script by running "./filename.py" in the terminal.

CHAPTER 4. MULTI-UAS ALGORITHM EXPERIMENT

In this chapter, we demonstrate the use of our multi-UAS testbed by implementing algorithm for (centralized) task allocation to heterogeneous agents.

4.1 Heterogeneous Agent Path Problem

The work of Prasad et al. [37] discusses the Heterogeneous Agent Path Problem (HAPP), derived from the multiple traveling salesperson (multi-TSP) problem, where the agents can be of different types (rather than homogeneous). HAPP is defined as follows. Consider a set of tasks T that are required to be completed by a set of k heterogeneous agents denoted as $A = \{A_1, A_2, ..., A_k\}$, where each agent is one of m types. Let $f : \{1, 2, ..., k\} \rightarrow \{1, 2, ..., m\}$ be a function that takes an agent number as input and outputs the type of that agent. For each $i \in \{1, 2, ..., m\}$, let m_i be the number of agents of type i, with $\sum_{i=1}^{m} m_i = k$. Let T be composed of two broad classes of tasks: type-specific tasks and generic tasks. Type specific tasks can only be completed by a specific type of agent, whereas generic tasks can be completed by any agent. Let T_0 denote the set of generic tasks and T_i, $i \in \{1, 2, ..., m\}$, denote the set of type-specific tasks that can be completed by agents of type i. Thus $T = T_0 \cup \{\cup_{i=1}^{m} T_i\}$. Each task has a corresponding location in the environment. Let v_j denote the start node of agent A_j. The set of all start nodes is denoted by $D = \{v_1, v_2, ..., v_k\}$. Let $G = (V, E)$ be the complete graph with vertex set $V = T \cup D$ (where D is the set of start locations of the agents) and edge set $E = \{(u, v) : u, v \in V, u \neq v\}$. Let each edge $e = (u, v) \in E$ have a weight $d(u, v)$ given by the distance between nodes u and v. A path is an alternating sequence of vertices and edges which begins and ends with vertices, such that all edges and vertices are distinct. The cost of the path is defined as the sum of the weights of the edges in the path.

Let S_j be the set of tasks allocated to agent A_j. Let $P_j^*(S_j)$ denote the cost of the optimal (lowest cost) path on the set $S_j \cup \{v_j\}$ starting from node v_j. There is no restrictions on the end point of the path. The goal is to divide the tasks T among agents subject to the task-agent compatibility constraints, such that the maximum path cost among all agents to visit all their allocated tasks from their respective start locations is minimized. Each task must be executed by exactly one agent and the task set allocated to each agent A_j is a union of type-specific tasks V_j (which is a subset of $T_{f(j)}$, where $f(j)$ is the type of agent A_j) and generic tasks R_j (which is a subset of T_0).

Thus, the Heterogeneous Agent Path Problem (HAPP) is defined as follows:

$$\begin{aligned} \min_{S_1,S_2,\ldots,S_k \subseteq T} \ & \max_{j\in[k]} \ P_j^*(S_j) \\ \text{subject to } & \cup_{j=1}^{k} S_j = T, \ S_j \cap S_i = \emptyset, \forall j \neq i \\ & S_j = V_j \cup R_j, \forall j \in [k], \\ & V_j \subseteq T_{f(j)}, R_j \subseteq T_0. \end{aligned}$$

4.2 The Hetero-Min-Max-Tree-Split Algorithm

The work of Prasad et al. [37] proposes a novel solution to HAPP task allocation involving agents of varying types and capabilities called the Hetero-Min-Max-Tree-Split algorithm. The input to this algorithm includes the number of agents of each type, a set of type specific tasks (with locations), a set of generic tasks (with locations), and the start locations of the agents. Given these inputs the algorithm proceeds in three phases to generate optimal paths for each agent. Phase one allocates type specific tasks by computing sub-trees on those tasks with start nodes for each agent. Phase two allocates generic tasks by balancing the total cost of travel for each agent by accounting for the allocation to agents after phase one. Lastly in phase three the type specific allocations and the generic task allocations are combined to create the final paths for each agent. These paths are comprised of the optimally arranged tasks (corresponding to locations) to be visited by each agent.

4.3 Multi-UAS HAPP Experiment

To implement and test the Hetero-Min-Max-Tree-Split algorithm in our testbed, we decided to use one Intel Aero drone, and one UVify Draco-R drone, and classified them as “type 1” and “type 2” agents respectively. We created 3 “type 1” specific tasks, 3 “type 2” tasks, and 1 generic task. The test area size was restricted to be 2 meters in the ‘x’ direction and 4 meters in the ‘y’ direction, centered at the origin of the flying space at x=0 and y=0. This means that all x coordinates generated must be between -1, and 1 meters, and all y coordinates generated must be between -2, and 2 meters. We also constrained the altitude at which both the Aero and Draco-R drones operated to be 1 meter and 2 meters, respectively. First, within the constraint of the test area, each agent was initiated at a random starting location. Then, the seven total task location were randomly generated within the test area.

Given these inputs, the Hetero-Min-Max-Tree-Split algorithm computed the optimal paths for each agent. We created a Python script in our custom ROS package to convert these paths into PoseStamped messages which contain a pose with reference coordinate frame and timestamp. This message is published on the “vehicle_ns/mavros/setpoint_position/local” rostopic for each vehicle. MAVROS converts these rostopic messages to PX4 hardware messages and sends the necessary flight commands for the drone to fly.

Figure 4.1 shows the resulting flight path according to the final paths generated for each agent according to the Hetero-Min-Max-Tree-Split algorithm. The red and blue points indicate the starting locations and tasks of the Intel Aero, and UVify Draco-R, respectively. Each point is numbered according to the index of the list of random way-points initially generated before being passed to the algorithm.

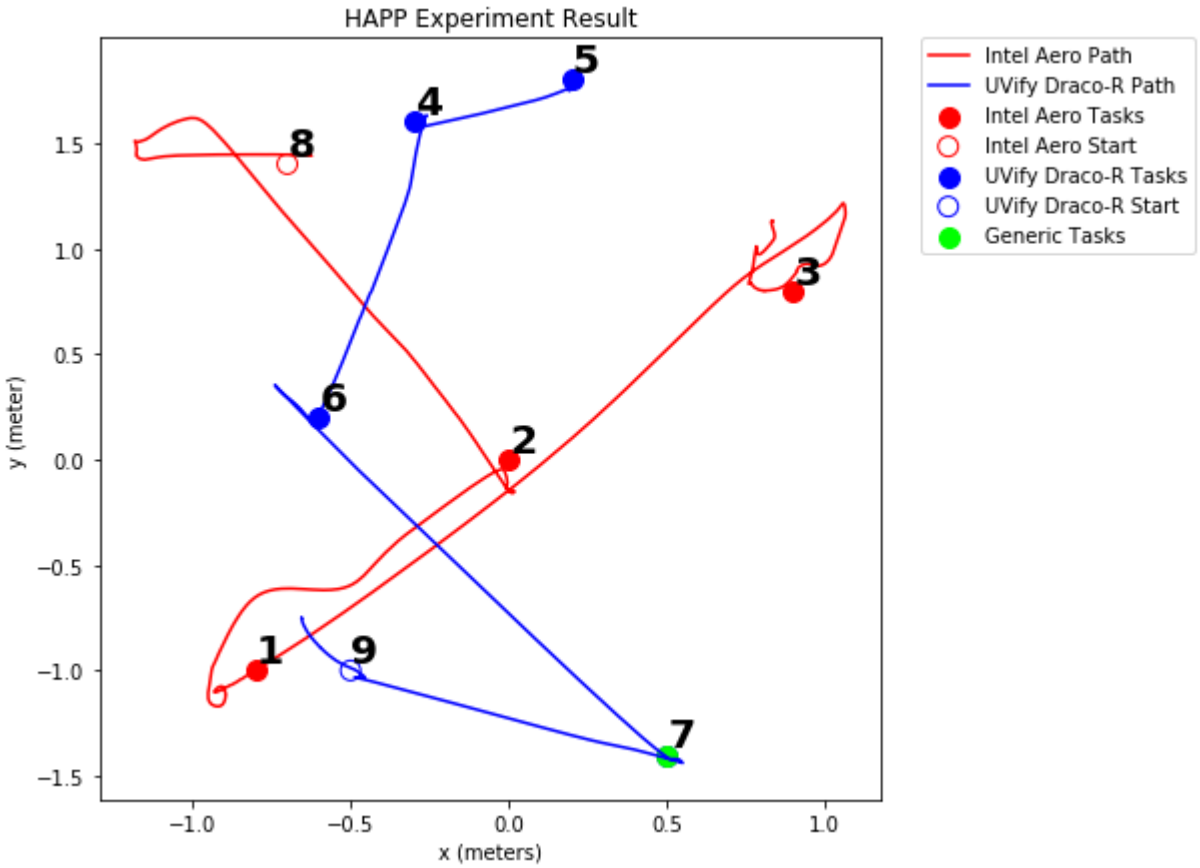

Figure 4.1. Flight Paths Arising from Task Allocation

CHAPTER 5. SUMMARY, CONCLUSION, AND RECOMMENDATIONS

In this thesis, we presented a compilation of information for establishing a multi-UAS laboratory. We offered insight into the obstacles faced and the solutions and pitfalls that we encountered which led to a fully operational multi-UAS laboratory. The biggest challenge that presented itself was hardware failure; when managing complex hardware it is not if hardware failure will happen, but when. Two unrelated instances of assumed flight controller hardware failure left us with two inoperable drones. Experiencing these failures however provided us with solutions that we, as well as other drone researchers, can apply in future work. Despite the aforementioned technical difficulty, the culmination of this work was the successful application of a multi-agent algorithm using two heterogeneous unmanned aerial systems. Another important conclusion we can draw following this work is the power and significance of open-source libraries. Such tools not only help get a project on its feet but propel drone research forward to new discoveries and implementations. We have contributed to this effort by creating a GitHub repository [10] that provides detailed instructions for setting up a multi-UAS laboratory, as well as source code. We hope that the findings of this work will be used to construct more multi-UAS laboratories that spur the development of intelligent UAS architectures. In the future we would like to repair and use the two defunct drones along with the two functioning drones to study new multi-agent algorithms.

REFERENCES

[1] M. R. Brust, G. Danoy, P. Bouvry, D. Gashi, H. Pathak, and M. P. Gonçalves. Defending against intrusion of malicious uavs with networked uav defense swarms. In *2017 IEEE 42nd Conference on Local Computer Networks Workshops (LCN Workshops)*, pages 103–111, 2017.

[2] D. Orfanus, E. P. de Freitas, and F. Eliassen. Self-organization as a supporting paradigm for military uav relay networks. *IEEE Communications Letters*, 20(4):804–807, 2016.

[3] M. Diehl, T. Klaproth, and M. Hotnung. System characteristics of a micro uav defense system. In *2019 IEEE International Systems Conference (SysCon)*, pages 1–8, 2019.

[4] D. Yadav, M. Choksi, and M. A. Zaveri. Supervised learning based greenery region detection using unnamed aerial vehicle for smart city application. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7, 2019.

[5] A. Vasudevan, D. A. Kumar, and N. S. Bhuvaneswari. Precision farming using unmanned aerial and ground vehicles. In *2016 IEEE Technological Innovations in ICT for Agriculture and Rural Development (TIAR)*, pages 146–150, 2016.

[6] P. Tokekar, J. Vander Hook, D. Mulla, and V. Isler. Sensor planning for a symbiotic uav and ugv system for precision agriculture. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5321–5326, 2013.

[7] Z. Zhou, C. Zhang, C. Xu, F. Xiong, Y. Zhang, and T. Umer. Energy-efficient industrial internet of uavs for power line inspection in smart grid. *IEEE Transactions on Industrial Informatics*, 14(6):2705–2714, 2018.

[8] S. Pan. Uav delivery planning based on k-means++ clustering and genetic algorithm. In *2019 5th International Conference on Control Science and Systems Engineering (ICCSSE)*, pages 14–18, 2019.

[9] S. Sawadsitang, D. Niyato, P. Tan, and P. Wang. Joint ground and aerial package delivery services: A stochastic optimization approach. *IEEE Transactions on Intelligent Transportation Systems*, 20(6):2241–2254, 2019.

[10] N. Coleman. Autonomous multi-uas lab setup, 2020. URL `https://github.com/nadiamcoleman/AUTONOMOUS-MULTI-UAS-LAB-SETUP`.

[11] Department of Computer and University of New Mexico Electrical Engineering. Marhes multi-agent robotics and heterogeneous systems laboratory, 2013. URL http://marhes.ece.unm.edu.

[12] Department of Aeronautics and Stanford University Astronautics. Multi-robot systems laboratory, 2012. URL https://msl.stanford.edu/.

[13] S. Wilson, P. Glotfelter, L. Wang, S. Mayya, G. Notomista, M. Mote, and M. Egerstedt. The robotarium: Globally impactful opportunities, challenges, and lessons learned in remote-access, distributed control of multirobot systems. *IEEE Control Systems Magazine*, 40(1):26–44, 2020.

[14] Open Source Robotics Foundation. Ros, . URL https://www.ros.org.

[15] Dronecode. Dronecode, . URL https://www.dronecode.org.

[16] Dronecode. Px4, . URL http://px4.io.

[17] Dronecode. Mavlink, . URL https://mavlink.io.

[18] Dronecode. Qgroundcontrol, . URL http://qgroundcontrol.com.

[19] ROS. Mavros, . URL http://qgroundcontrol.com.

[20] Open Source Robotics Foundation. Ros, . URL http://gazebosim.org.

[21] S. Ivaldi, J. Peters, V. Padois, and F. Nori. Tools for simulating humanoid robot dynamics: A survey based on user feedback. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 842–849, 2014.

[22] College of Engineering, Purdue. Facilities / current assets, 2018. URL https://engineering.purdue.edu/Initiatives/AutoSystems/Facilities. [Online; accessed April 21, 2020].

[23] Dronecode. Qgroundcontrol with px4, . URL https://discuss.px4.io/c/qgroundcontrol/qgroundcontrol-usage.

[24] Intel. Intel® aero products, 2018. URL https://www.intel.com/content/www/us/en/support/products/97174/drones/development-drones/intel-aero-products.html. [Online; accessed April 24, 2020].

[25] UVify. Draco-r, 2019. URL https://www.uvify.com/draco-r/. [Online; accessed April 24, 2020].

[26] Dronecode. Intel aero ready to fly drone, 2018. URL https://docs.px4.io/v1.9.0/en/complete_vehicles/intel_aero.html. [Online; accessed April 24, 2020].

[27] Yun Wei. 02 initial setup, May 2018. URL https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-Setup.

[28] Paul Guermonprez. 90 (references) os user installation, Oct 2018. URL https://github.com/intel-aero/meta-intel-aero/wiki/90-(References)-OS-user-Installation.

[29] Open Source Robotics Foundation. Ros, . URL http://wiki.ros.org/ROS/Installation.

[30] UVify. Draco-r, 2019. URL https://store.uvify.com/collections/draco-r/products/draco-r. [Online; accessed April 24, 2020].

[31] Intel. Mavlink router. URL https://github.com/intel/mavlink-router.

[32] Dronecode. Parameter reference, . URL https://dev.px4.io/v1.9.0/en/advanced/parameter_reference.html.

[33] ROS. Ros packages, . URL http://docs.ros.org/independent/api/rospkg/html/packages.html.

[34] Dronecode. Multi-vehicle simulation with gazebo, . URL https://dev.px4.io/v1.9.0/en/simulation/multi-vehicle-simulation.html.

[35] Dronecode. Simulation, . URL https://dev.px4.io/master/en/simulation/.

[36] Dronecode. Ros with gazebo simulation, 2018. URL https://dev.px4.io/v1.9.0/en/simulation/ros_interface.html. [Online; accessed April 8, 2020].

[37] A. Prasad, H. Choi, and S. Sundaram. Min-max tours and paths for task allocation to heterogeneous agents. *IEEE Transactions on Control of Network Systems*, pages 1–1, 2020.